(12) NACH DEM VERTRAG ÜBER DIE INTERNATIONALE ZUSAMMENARBEIT AUF DEM GEBIET DES PATENTWESENS (PCT) VERÖFFENTLICHTE INTERNATIONALE ANMELDUNG

(19) Weltorganisation für geistiges Eigentum
Internationales Büro

551454

(43) Internationales Veröffentlichungsdatum
14. Oktober 2004 (14.10.2004)

PCT

(10) Internationale Veröffentlichungsnummer
WO 2004/088549 A2

(51) Internationale Patentklassifikation[7]: G06F 17/50

(21) Internationales Aktenzeichen: PCT/EP2004/003301

(22) Internationales Anmeldedatum:
29. März 2004 (29.03.2004)

(25) Einreichungssprache: Deutsch

(26) Veröffentlichungssprache: Deutsch

(30) Angaben zur Priorität:

10314834.5	1. April 2003 (01.04.2003)	DE
10314835.3	1. April 2003 (01.04.2003)	DE
10314831.0	1. April 2003 (01.04.2003)	DE
10314832.9	1. April 2003 (01.04.2003)	DE

(71) Anmelder *(für alle Bestimmungsstaaten mit Ausnahme von US)*: **SIEMENS AKTIENGESELLSCHAFT** [DE/DE]; Wittelsbacherplatz 2, 80333 München (DE).

(72) Erfinder; und

(75) Erfinder/Anmelder *(nur für US)*: **OBERHAUSER, Roy** [DE/DE]; Egmatinger Str. 9a, 85667 Oberpframmern (DE). **REICHEL, Christian** [DE/DE]; Pestalozzistr. 13, 01307 Dresden (DE).

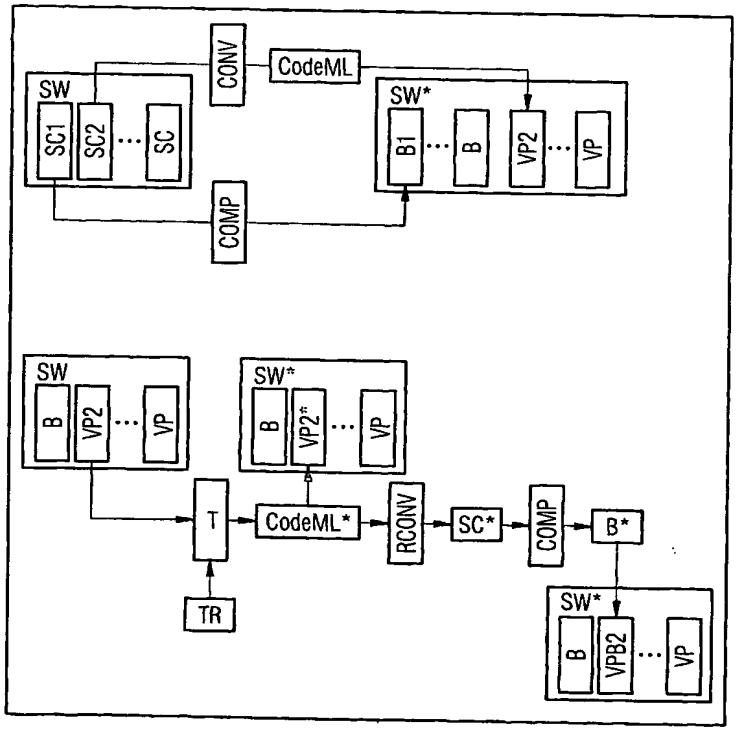
(74) Gemeinsamer Vertreter: **SIEMENS AKTIENGESELLSCHAFT**; Postfach 22 16 34, 80506 München (DE).

(81) Bestimmungsstaaten *(soweit nicht anders angegeben, für jede verfügbare nationale Schutzrechtsart)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM,

[Fortsetzung auf der nächsten Seite]

(54) Title: METHOD AND ARRAY FOR CHANGING SOFTWARE OR SOURCE CODE

(54) Bezeichnung: VERFAHREN UND ANORDNUNG ZUR VERÄNDERUNG VON SOFTWARE ODER QUELLCODE



program content or functionality has been changed.

(57) Abstract: The invention basically relates to the following: in a first variant, selected components of a software are used as variation points, wherein said components are transformed into a first XML code. The software, which is now in mixed form, is delivered. On the user side, the first code is converted into a second XML code depending only upon transformation rules by means of one or more transformations, e.g. XSLT. In a second variant, a first XML code, which contains at least one language extension, is converted into a second, more easily verifiable XML code without said language extensions depending on transformation rules. In a third variant, a source code formulated in XML is transformed in such a way that a new source code is obtained after back-conversion into the original programming language, in which not only the presentation but also the actual program content or functionality has been changed. In a fourth variant, a source code formulated in XML, for instance, with initial states, code fragments to be exchanged and foreign language modules tailored to the corresponding natural language of the user, are mixed by means of transformation, whereby a new source code is obtained after back-conversion, in which not only the presentation but also the actual

[Fortsetzung auf der nächsten Seite]

WO 2004/088549 A2

TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) **Bestimmungsstaaten** *(soweit nicht anders angegeben, für jede verfügbare regionale Schutzrechtsart)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), eurasisches (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), europäisches (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Veröffentlicht:

— *ohne internationalen Recherchenbericht und erneut zu veröffentlichen nach Erhalt des Berichts*

Zur Erklärung der Zweibuchstaben-Codes und der anderen Abkürzungen wird auf die Erklärungen ("Guidance Notes on Codes and Abbreviations") am Anfang jeder regulären Ausgabe der PCT-Gazette verwiesen.

(57) **Zusammenfassung:** Die Erfindung besteht im Wesentlichen darin, dass in einer ersten Variante ausgewählte Bestandteile einer Software als Variationspunkte dienen, indem diese in einen in einen ersten XML-Code umgewandelt werden, die Software nun in Mischform, ausgeliefert wird, und der erste Code kundenseitig durch eine oder mehrere Transformationen, bspw. XSLT, nur in Abhängigkeit von Transformationsregeln in einen in einen zweiten XML-Code umgewandelt werden, dass in einer zweiten Variante ein erster XML-Code, der mindestens eine Spracherweiterung enthält in Abhängigkeit von Transformationsregeln in einen leichter verifizierbaren zweiten XML-Code ohne diese Spracherweiterungen überführt wird, dass in einer dritten Variante ein in XML formulierter Quellcode derart transformiert wird, dass, nach einer Rückkonvertierung in die ursprüngliche Programmiersprache, ein neuer Quellcode entsteht, bei dem nicht nur die Darstellung, sondern auch der eigentliche Programminhalt bzw. die Funktionalität verändert wurde, oder dass in einer vierten Variante ein in XML formulierter Quellcode mit beispielsweise Ausgangszuständen, auszutauschenden Code-Fragmenten und auf die jeweilige natürliche Sprache des Anwenders zugeschnittenen Fremdsprachenmodulen durch Transformation vermischt wird, wodurch, nach einer Rückkonvertierung ein neuer Quellcode entsteht, bei dem nicht nur die Darstellung, sondern auch der eigentliche Programminhalt bzw. die Funktionalität verändert wurde.

Beschreibung

Verfahren und Anordnung zur Veränderung von Software oder Quellcode

Die Erfindung betrifft ein Verfahren und eine Anordnung zur Veränderung von Software oder Quellcode, bei dem/der eine Software bzw. ein Quellcode, in eine Darstellung in einer Meta-Auszeichnungssprache, beispielsweise XML, übergeführt, dort, beispielsweise mit XSLT, transformiert und dann diese in der Meta-Auszeichnungssprache formulierte transformierte Darstellung in eine modifizierte Software oder in einen modifizierten Quellcode, beispielsweise derselben Ausgangssprache, zurückverwandelt wird.

Aus dem Stand der Technik sind zwar einige Möglichkeiten zur nachträglichen Veränderung bzw. Modifikationen von Software bekannt, die jedoch alle einige Nachteile gegenüber der Erfindung aufweisen:

Eine Möglichkeit der Einflussnahme auf Software erfolgt mit Hilfe einer Parametrisierung. Konfigurationsdateien werden typischerweise zur Parametrisierung und Speicherung anwendungsspezifischer „Parameterdaten" verwendet. Die Struktur und der Aufbau dieser Dateien wird dabei während der Entwicklungsphase festgelegt, und ist in keinem Fall nach der Auslieferung der Software veränderbar.

PlugIns eröffnen die Möglichkeit schon „ausgelieferte", kompilierte Software um Funktionalitätseigenschaften zu erweitern. Hierfür ist es notwendig, dass die Strukturen zur Einbindung und Nutzung von PlugIns bereits während der Entwicklungsphase erstellt bzw. festgelegt werden (Schnittstellen,...).

Code-Generatoren erzeugen Quell- oder BinärCode mit Hilfe von Templates/Vorlagen, die an vorgegebenen Stellen, z. B.

mittels übergebener Parameter, vervollständigt werden. Auf diese Weise wird es möglich, dass z. B. für verschiede Kunden unterschiedliche Software erzeugt wird, die sich an genau definierten Stellen unterscheidet. Hierbei sind aber nur spezielle Stellen (und nicht beliebige Stellen) im Code veränderbar, welche mit der Erstellung des Templates genau spezifiziert werden müssen. Code-Generatoren werden typischer Weise auf Entwicklerseite eingesetzt.

US-Patentanmeldung US 6052531A1 ist eine spezielle Anwendungsmöglichkeit von Variationspunkten in Form von Updates/Patches bekannt.

Aus dem Internet ist unter http://beautyj.berlios.de/ ein Java Source Code Transformation Tool BeautyJ bekannt, bei dem ein Java Quellcode in eine XML-Darstellung umgewandelt wird, mittels Sourclet API, beispielsweise durch Einfügen von Leerzeichen oder geänderten Kommentaren an bestimmten Stellen, „verschönert" und anschließend der modifizierte Quellcode in Java Quellcode zurück konvertiert werden kann. Eine Transformation mittels XSLT wird hier, für diesen Zweck, nur vorgeschlagen, aber nicht umgesetzt.

Die der Erfindung zugrunde liegende Aufgabe liegt nun darin, ein Verfahren und eine Anordnung zur Modifikation von Quellcode anzugeben, bei dem/der eine weitergehende noch flexiblere und effizientere Modifikation der Software bzw. des Quellcodes erreicht wird.

Diese Aufgabe wird hinsichtlich des Verfahrens durch die Merkmale der Patentansprüche 1, 6, 8 und 13 und hinsichtlich der Anordnung durch die Merkmale der Patentansprüche 22 bis 26 erfindungsgemäß gelöst. Die weiteren Ansprüche betreffen bevorzugte Ausgestaltungen der Erfindung.

Die Erfindung besteht im Wesentlichen darin,

dass in einer ersten Variante ausgewählte Bestandteile einer Software als Variationspunkte dienen, indem diese in einen in einer Meta-Auszeichnungssprache, bspw. XLML, formulierten ersten Code umgewandelt werden, die Software nun in Mischform, ausgeliefert wird, und der erste Code kundenseitig durch eine oder mehrere Transformationen, bspw. XSLT, nur in Abhängigkeit von Transformationsregeln in einen in der Meta-Auszeichnungssprache formulierten zweiten Code umgewandelt werden,

dass in einer zweiten Variante ein in einer Meta-Auszeichnungssprache formulierter erster Code, der mindestens eine Spracherweiterung enthält in Abhängigkeit von Transformationsregeln in einen in der Meta-Auszeichnungssprache formulierten leichter verifizierbaren zweiten Code ohne diese Spracherweiterungen überführt wird,

dass in einer dritten Variante ein in eine Meta-Auszeichnungssprache transformierter Quellcode derart transformiert wird, dass, nach einer Rückkonvertierung in die ursprüngliche Programmiersprache, ein neuer Quellcode entsteht, bei dem nicht nur die Darstellung, sondern auch der eigentliche Programminhalt bzw. die Funktionalität verändert wurde, oder

dass in einer vierten Variante ein in eine Meta-Auszeichnungssprache transformierter Quellcode mit beispielsweise Ausgangszuständen, auszutauschenden Code-Fragmenten und auf die jeweilige natürliche Sprache des Anwenders zugeschnittenen Fremdsprachenmodulen durch Transformation vermischt wird, wodurch, nach einer Rückkonvertierung ein neuer Quellcode entsteht, bei dem nicht nur die Darstellung, sondern auch der eigentliche Programminhalt bzw. die Funktionalität verändert wurde.

Die Erfindung wird im Folgenden anhand der in den Zeichnungen dargestellten Beispiele näher erläutert. Dabei zeigt

Figur 1 ein Gesamtblockdiagramm zur Erläuterung einer ersten Erfindungsvariante,

Figur 2 ein Gesamtblockdiagramm zur Erläuterung einer zweiten Erfindungsvariante,

Figur 3 ein Blockschaltbild zur Erläuterung der erfindungsgemäßen Überführung von PreProcessing-Erweiterungen,

Figur 4 ein Gesamtblockdiagramm zur Erläuterung einer dritten Erfindungsvariante,

Figur 5 ein Blockschaltbild zur Erläuterung der erfindungsgemäßen Modifikation durch die Verwendung von Aspekten,

Figur 6 ein Blockschaltbild zur Erläuterung des erfindungsgemäßen Einfügens von Migrierbarkeits-Funktionalität,

Figur 7 ein Blockschaltbild zur Erläuterung der erfindungsgemäßen Modifikation durch den Einsatz von Templates, Filtern und Patterns,

Figur 8 Gesamtblockdiagramm zur Erläuterung einer vierten Erfindungsvariante,

Figur 9 ein Blockschaltbild zur Erläuterung des erfindungsgemäßen Austauschens von Code-Fragmenten,

Figur 10 ein Blockschaltbild zur Erläuterung des erfindungsgemäßen Einfügens von Zustandsdaten,

Figur 11 ein Blockschaltbild zur Erläuterung der Variationsmöglichkeiten des erfindungsgemäßen Einbaus von Informationen und

Figur 12 ein Blockschaltbild zur Erläuterung des erfindungsgemäßen Einbaus von Fremdsprachmodulen zur Internationalisierung des Quellcodes.

1. Erfindungsvariante

In **Figur 1** ist ein Gesamtblockdiagramm zur Erläuterung der Erfindung dargestellt, bei dem zunächst eine Software **SW** bestehend aus Quelltext **SC1**, **SC2** und **SC** in eine auslieferungsfähige Software **SW*** umgewandelt wird, wobei einige Teile der Software wie bspw. **SC1** nun als Binärcode/Byte Code **B1** zur Verfügung stehen, und andere Teile wie bspw. **SC2** durch einen Konverter **CONV** in einen in einer Meta-Auszeichnungssprache formulierten ersten Code **CodeML** umgewandelt werden, so daß sie in der lauffähigen Software **SW*** fortan Variationspunkte **VP** bspw. **VP1** bilden. Diese Software **SW*** kann vor/oder zur Laufzeit auf eine Weise modifiziert werden, das der in der Meta-Auszeichnungssprache dargestellte Code **VP**, bspw. **VP2**, mit einer Transformation **T** und Transformationsregeln **TR** in einen zweiten in der Meta-Auszeichnungssprache formulierten Code **CodeML*** umgewandelt wird, welcher nun entweder als geänderter Variationspunkt bspw. **VP2*** in **SW*** vorhanden ist oder durch einen Konverter **RCONV** nach der Transformation **T** in einen Quellcode **SC*** und danach mittels **COMP** in einen ByteCode/Binärcode **VP2B*** überführt wird. In beiden Fällen unterscheiden sich **SW** und **SW*** an den Stellen der Variationspunkte und können auf diese Weise an spezifische Anforderungen (bspw. Tookit-Austausch, Updates, usw.) angepasst werden.

Die Codes **CodeML** und **CodeML*** bzw. **VP** und **VP*** sind beispielsweise in der Meta-Auszeichnungssprache XML formuliert, wobei „XML" für Extended Markup Language steht.

Von besonderem Vorteil ist hierbei, dass dies nicht vom Programmentwickler durchgeführt werden muss, sondern vom

entsprechend ausgestatteten und informierten Kunden selbst erledigt werden kann. Hierzu braucht ein Operator oder Administrator auf der Kundenseite nur eine entsprechende Transformation **T** mit den benötigten Ersetzungs-, Modifikations- und Entfernungsregeln **TR** anzuwenden, um die Software auf seine speziellen Bedürfnisse anzupassen bzw. ein Update oder Patching durchzuführen. Beim Update oder Patching von kundenspezifisch angepasster treten bislang häufig Probleme wegen Inkonsistenzen auf, die durch diese Erfindung und die Möglichkeit der Pipelineanwendung bzw. geordnete Hintereinanderausführung von Anfang an vermieden werden können.

Die im **Anhang 1** befindlichen Programmauflistungen **Listing 1** bis **Listing 5** zeigen dies an einem konkreten Beispiel: Typischerweise kann eine Software-Auslieferung an zwei unterschiedliche Kunden unterschiedliche Toolkits verwenden, die sich hinsichtlich Performance, Preis usw. unterscheiden.

So soll hier ein Code der ursprünglich eine Registrierungsklasse
import electric.registry.Registry
aus einem **Glue-Toolkit** verwendet, beim zweiten Kunden nun zwei neue "Registrierungsklassen"
import org.apache.axis.client.Call und
import org.apache.axis.client.Service aus einem **Axis-Tookit** verwenden.

In XSL kann dies z.B. mittels

```
<xsl:template match="import">
    <xsl:if test="dot/name='Registry'">
     <import>
       <dot>
         <dot><dot><dot><name>org</name><name>apache</name></dot><name>axis</name></dot>
         <name>client</name></dot><name>Call</name>
       </dot>
     </import>
     <import>
       <dot>
         <dot><dot><dot><name>org</name><name>apache</name></dot><name>axis</name></dot>
           <name>client</name></dot><name>Service</name>
       </dot>
     </import>
    </xsl:if>
```

```
    <xsl:if test="dot/name!='Registry'">
      <xsl:copy-of select="."/>
    </xsl:if>
    ...
  </xsl:template>
```

geschehen. Schablonen bzw. Templates werden in XSL auf das in **match** definierte Muster angewendet. Das import-Template im Listing-Beispiel also auf alle ursprünglichen import-Anweisungen. Im konkreten Beispiel ignoriert es einfach alle ursprünglichen GlUE-Registry-Imports, und fügt stattdessen die zwei Axis-spezifischen imports ein.

Durch das erfindungsgemäße Verfahren ergeben sich noch eine Reihe von zusätzlichen Vorteilen, wie beispielsweise:

1. Es ist nur ein System für Problemstellungen wie Patching, Customizing, Updating, etc. erforderlich und nicht eine Reihe verschiedener teilweise proprietärer Werkzeuge.

2. Das Verfahren basiert auf Standards wie XML und XSLT und ist hinsichtlich der Konvertierbarkeit in andere Programmiersprachen geringeren Beschränkungen unterworfen als andere Verfahren zur Modifikation von Quellcode.

3. Selbst für spezielle und komplizierte Quellcode-Modifikationen sind keine proprietären Speziallösungen erforderlich, sondern es können hierfür existierende Standards wie **XSLT**, **XPath** und **XQuery** genutzt werden.

4. Diese Art der Modifikation erlaubt die Erstellung von Hierarchien u.a. durch die Möglichkeit zur geordneten, automatisierten Hintereinanderausführung (Pipelines) mehrerer Transformationen, bspw. von Patches.

5. Die Transformationen können für eine allgemeine Wiederverwendung in XSLT-Dateien gespeichert werden, so daß Bibliotheken z.B. für bestimmte Abläufe entstehen können.

6. Es kann eine XML-Repräsentation des Quellcodes in einer XML-Datenbasis gespeichert und bei Bedarf mit Hilfe einer XSLT in einfacher Weise an die jeweiligen Kundenbedürfnisse angepasst werden (Customization).

7. Durch die Verwendung der Standards **XMLSchema** oder **DTD** oder entsprechende XSLTs kann der Code vorab (ohne Kompilierung), auf bestimmte Korrektheitsaspekte hin, überprüft (validiert) werden.

8. Übliche XML-Tools können zur einfachen Bearbeitung bzw. Visualisierung und Bestimmung von Beziehungen im Code verwendet werden.

9. Dauerhafte XML-basierte Programmbibliotheken, die XPath-Anfragen unterstützen, können die Wiederverwendung von Code durch besseres Auffinden eines Codes bzw. von Code-Fragmenten oder Templates verbessert werden.

2. Erfindungsvariante

In **Figur 2** ist ein Gesamtblockdiagramm zur Erläuterung der Erfindung dargestellt, bei dem ein in einer Meta-Auszeichnungsprache formulierter erster Code **CodeML**, der eine Spracherweiterung **LE** enthält und durch **RCONV** nicht in gültigen Quelltext **SC*** überführbar ist, durch eine Transformation **T** in Abhängigkeit von Transformationsregeln **TR**, welche einen Sprachkonverter **LC** enthalten, in einen in der Meta-Auszeichnungssprache formulierten zweiter Code **CodeML*** ohne überführt wird, welcher keine diese Spracherweiterungen **LE** enthält und deshalb in einen Quellcode **SC*** verwandelt werden kann, welcher mittels eines Compilers **COMP** wiederum in gültigen Binärcode/ByteCode **B*** überführbar ist.

Der modifizierte Quellcode **SC*** sind beispielsweise in der Programmiersprache Java und die Codes **CodeML** und **CodeML*** sind beispielsweise in der Meta-Auszeichnungssprache XML formuliert, wobei „XML" für Extended Markup Language steht.

Die Transformation **T**, z. B. eine Extended Stylesheet Language Transformation oder **XSLT**, wird durch Transformationsregeln **TR**, z. B. innerhalb von **XSL** (Extended Stylesheet Language) Dateien beschrieben, wobei bspw. die in XSL formulierten Regeln u.a. als Language Converter LC verwendet werden und beschreiben wie der in XML-codierte Quellcode **CodeML** mit einer Spracherweiterung **LE** in eine Variante ohne Spracherweiterung transformierbar ist.

In **Figur 3** ist ein erstes Ausführungsbeispiel dargestellt, bei dem ein in einer Meta-Auszeichnungssprache formulierter erster Code **CodeML** eine Spracherweiterung für PreProcessing **PPE** (z.B. <define>, <ifdef>, usw.) enthält, und mit Hilfe einer Transformtion **T** in Abhängigkeit von Transformationsregeln **TR**, welche einen PreProcessing Language Converter **PPLC** besitzen der **PPE** auflöst bzw. anwendet, in einen in der Meta-Auszeichnungssprache formulierten zweiten Code CodeML* ohne Spracherweiterung transformiert wird.

Die Ausgestaltung der Spracherweiterung erfolgt typischerweise in Form von Elementen für die generische Programmierung **1** , und/oder für PreProcessing **2** und/oder eine kunden- bzw. entwicklerspezifische Grammatik **3** und/oder für Makros **4**.

Alle Spracherweiterung bzw. der CodeML selbst kann jederzeit durch den Einsatz von DTDs(document type definitions) bzw. XMLSchema validiert werden.

Der Programmierer erhält durch die Erfindung mehr Freiheiten, da die Grammatik der verwendeten Programmiersprache auf die

Wünsche des Programmierers abgestimmt werden kann und eine Rücktransformation auf die normale Grammatik der Programmiersprache erst am Ende der Programmentwicklung erfolgen muss. Ein wesentlicher Vorteil besteht auch darin, dass eine Validierung der Spracherweiterungen mit einem für die normale Programmiersprache vorgesehenen Compiler erfolgen kann.

Die im **Anhang 2** befindlichen Programmauflistungen **Listing 1** bis **Listing 3** zeigen die Auflösung der PreProcessing-Erweiterungen **PPE** an einem konkreten Beispiel, bei dem die in **Listing 1** dargestellte Klasse **TestOutput.xjava** eine **PPE** in Form von <define name="m" value="private"> enthält, welche Einfluß auf die Werte der <m>-Elemente nimmt, und durch eine Transformation **T** in Abhängigkeit von Transformationsregeln **TR** (hier: **PPLC**) nun in die in **Listing 2** dargestellte XML-basierte Form **TestOutput.xjava*** überführt wird, bei der alle <m>-Elemente durch ein über value="private" bestimmtes <private/>-Element ersetzt werden. Hiermit wird es möglich **TestOutput.xjava*** in den in **Listing 3** aufgezeigten Quellcode **TestOutput.java** zu überführen.

Durch das erfindungsgemäße Verfahren ergibt sich noch eine Reihe von weiteren Vorteilen, wie beispielsweise:

1. Es ist nur ein System für Problemstellungen wie Customizing von Programmiersprachen, usw. erforderlich und nicht eine Reihe verschiedener teilweise proprietärer Werkzeuge.

2. Das Verfahren basiert auf Standards wie XML und XSLT und ist hinsichtlich der Konvertierbarkeit in andere Programmiersprachen geringeren Beschränkungen unterworfen als andere Verfahren zur Modifikation von Quellcode.

3. Selbst für spezielle und komplizierte Quellcode-Modifikationen sind keine proprietären Speziallösungen

erforderlich, sondern es können hierfür existierende Standards wie **XSLT, XPath** und **XQuery** genutzt werden.

4. Diese Art der Modifikation erlaubt die Erstellung von Hierarchien u.a. durch die Möglichkeit zur geordneten, automatisierten Hintereinanderausführung (Pipelines) mehrerer Transformationen, bspw. von Sprachanpassungen.

5. Die Transformationen können für eine allgemeine Wiederverwendung in XSLT-Dateien gespeichert werden, so daß Bibliotheken z.B. für bestimmte Abläufe entstehen können.

6. Es kann eine XML-Repräsentation des Quellcodes in einer XML-Datenbasis gespeichert und bei Bedarf mit Hilfe einer XSTL in einfacher Weise an die jeweiligen Kunden- bzw. Entwicklerbedürfnisse angepasst werden (Customization).

7. Durch die Verwendung der Standards **XMLSchema** oder **DTD** oder entsprechende XSLTs kann der Code vorab (ohne Kompilierung), auf bestimmte Korrektheitsaspekte hin, überprüft (validiert) werden.

8. Übliche XML-Tools können zur einfachen Bearbeitung bzw. Visualisierung und Bestimmung von Beziehungen im Code verwendet werden.

3. Erfindungsvariante

In **Figur 4** ist ein Gesamtblockdiagramm zur Erläuterung der Erfindung dargestellt, bei dem zunächst ein Quellcode **SC** durch einen Konverter **CONV** in einen in einer Meta-Auszeichnungssprache formulierten ersten Code **CodeML** umwandelt wird, wobei der Quellcode **SC** bei sofortiger Kompilierung einen Byte Code bzw. Binärcode **B** ergeben würde. Der in der Meta-Auszeichnungssprache dargestellte Code **CodeML**

wird nun auf dem Wege einer Transformation **T** ausschließlich durch die Verwendung von Transformationsregeln **TR** modifiziert, welche aus Bedingungen **C** und/oder Logik **L** und/oder Code-Fragmenten **CF** bestehen, wodurch sich ein zweiter ebenfalls in der Meta-Auszeichnungssprache formulierten Code **CodeML*** ergibt. Ein weiterer Konverter **RCONV** wandelt nach der Transformation den Code **CodeML*** in einen Quellcode **SC*** zurück, der typischerweise in derselben Sprache wie der Quellcode **SC** formuliert ist. Durch einen Compiler **COMP** wird schließlich der modifizierte Code **SC*** in einen modifizierten Byte Code **B*** oder aber gleich in einen ausführbaren Binärcode umgewandelt. Wesentlich ist hierbei, dass sich der Byte-Code **B*** prinzipiell vom Byte-Code **B** unterscheidet bzw. dass der Quellcode nicht nur in seiner Darstellung, sondern auch in seinem Programmablauf geändert wurde.

Der Quellcode **SC** und der modifizierte Quellcode **SC*** sind beispielsweise in der Programmiersprache Java und die Codes **CodeML** und **CodeML*** sind beispielsweise in der Meta-Auszeichnungssprache XML formuliert, wobei „XML" für Extended Markup Language steht.

Die Transformation T, z. B. eine Extended Stylesheet Language Transformation oder **XSLT**, wird durch Transformationsregeln TR, z. B. innerhalb von **XSL** (Extended Stylesheet Language) Dateien beschrieben, wobei bspw. die in XSL formulierten Regeln **TR** u.a. beschreiben wie der in XML-codierte Quellcode **CodeML** mit dem Code-Fragment **CF** kombiniert wird, um einen neuen modifizierten Quellcode **CodeML*** mit integriertem **CF**, oder eine Abwandlung davon, zu bilden, welcher nun beispielsweise zusätzliche Logging-Funktionalität enthalten kann.

In **Figur 5** ist ein erstes Ausführungsbeispiel dargestellt, bei dem die Transformationsregeln **TR** speziell Aspektregeln **AR** nach Aspektorientierter Programmierung (AOP) entsprechen, die

in der AspectJ-Sprache ausgedrückt mindestens einen Pointcut **PC** und/oder mindestens einen Advice-Type **AT** und/oder mindestens ein Advice-Body **AB** enthalten und in ihrer Reihenfolge den Bestandteilen aus **Figure 5** zugeordnet werden können.

Auf diese Weise kann eine (werkzeugunabhängige) sogenannte AOP realisiert werden, die im Vergleich zu anderen Lösungsvarianten, beispielsweise AspectJ, keinen zusätzlichen Overhead im generierten Code **CodeML*** erzeugt und nicht den üblichen Einschränkungen (extra Kompiler, Syntax, usw.) existierender Aspektsprachen unterliegt.

Als Aspekt bezeichnet man in AOP eine Einheit, die überkreuzende Anliegen(crosscutting concerns) z.B. ein Logging, modularisiert und an einer Stelle kapselt. Der entsprechende Code, der bisher mehrere Module durchzog, wird hierbei mit Hilfe eines einzigen Aspekts zusammengeführt.

Die im **Anhang 3** befindlichen Programmauflistungen **Listing 1** bis **Listing 5** zeigen dies an einem konkreten Beispiel, bei dem zunächst die in Listing 1 enthaltene Datei TestCalculator.java in eine XML-Darstellung TestCalculator.xjava konvertiert wird. In **Listing 3** erfolgt die Beschreibung eines Aspektes in Form einer Datei LoggingAspect.xsl, die alle notwendigen Transformationsregeln enthält und dafür sorgt, dass jede Methode, die ein „cal" in ihrem Namen trägt, gefunden wird und zu Beginn der Ausführung dieser Methode ein Druckbefehl **System.out.println(„calculate begin")** und am Ende der Ausführung dieser Methode ein Druckbefehl **System.out.println(„calculate end")** eingesetzt wird.

Sollen z.B. in allen 151 Klassen eines Projektes, alle Methoden die dem Muster **„cal"** entsprechen, also z.B. public String calcValues() o. ä., dazu veranlasst werden, eine

System-Ausgabe beim Eintritt und Austritt zu tätigen, so werden zunächst mit

```
match="*[(name()='curly')and(ancestor::method[contains(name,'cal')])]"
```

alle Methoden mit dem „cal"- Muster ausgewählt, mit

```
<expr>
 <paren>
  <dot><dot><name>System</name><name>out</name></dot><name>println</name></dot>
  <exprs>
   <expr>
    <xsl:text>"</xsl:text><xsl:value-of select="../name"/><xsl:text> begin"</xsl:text>
   </expr>
  </exprs>
 </paren>
</expr>
```

ein Statement **"System.out.println(%Name der Methode% + " begin")"**, z.B. **System.out.println("calculate end")**, eingefügt, mit

```
<xsl:copy-of select="*" />
```

der ursprünglichen Code der Methode eingefügt und mit

```
<expr>
 <paren>
  <dot><dot><name>System</name><name>out</name></dot><name>println</name></dot>
  <exprs>
   <expr>
    <xsl:text>"</xsl:text><xsl:value-of select="../name"/><xsl:text> end"</xsl:text>
   </expr>
  </exprs>
 </paren>
</expr>
```

ein Statement **"System.out.println(%Name der Methode% + " end")"**, z.B. **System.out.println("calculate end")** eingefügt.

Anstatt also in allen 151 Klassen eine entsprechende Logging-Ausgabe zu veranlassen, kann dies hier innerhalb eines Logging-Aspektes an einer Stelle erfolgen. Änderungen müssen so auch nur an einer Stelle vorgenommen werden.

Figur 6 betrifft ein zweites Anwendungsbeispiel der Erfindung, bei dem ebenfalls aus einem Quellcode **CodeML** durch

die Transformation **T** ein transformierter Code **CodeML*** erzeugt wird, welcher nun einen Mechanismus zur Sicherung (OLD) bzw. Ermittlung (NEW) mindestens eines Zustandes für die gewünschte (Versions)Migration enthält. Die Transformationsregeln **TR** sind in diesem Fall derart ausgebildet, dass sie als Migrationsregeln **MR** bezeichnet werden können und neben **C** und **L** zusätzlich mind. ein Fragment, sogenannte Checkpoints **CP**, zur Generierung (**CP Write**) bzw. zum Einlesen (**CP Read**) von Zuständen (**CP Data**) enthalten, die eine Migration von einer älteren Version **B*OLD** auf eine neuere Version **B*NEW** ermöglichen.
Die für eine Migration erforderliche Formatkonvertierungen der zu übergebenden Systemzustände können hiermit ebenfalls berücksichtigt werden. Zukünftige Migrationen brauchen hierdurch nicht schon im Vorfeld berücksichtigt zu werden, wodurch der Testaufwand und diesbezügliche potenzielle Programmfehler in frühen Programmversionen vermieden werden. Durch eine Automatisierung der Migration werden menschliche Fehler vermieden, da die Migration wesentlich systematischer erfolgt.

In **Figur 7** ist ein drittes Ausführungsbeispiel in mehreren Untervarianten dargestellt, bei dem ebenfalls ein in XML-codierter Quellcode **CodeML** durch eine Transformation **T** in einen modifizierten **CodeML*** umgewandelt wird. Die Transformation **T** wird hier jedoch durch Transformationsregeln **TR** bewirkt, die in jeder Variante aus mindestens **C** und **L** bestehen und wie bei der Umsetzung von Templates **TP** zusätzlich mindestens ein Template-Fragment **TPF** enthalten, bspw. für die Umwandlung in eine **EJB** (Enterprise Java Bean) und bei der Umsetzung von Patterns **P** mindestens ein Pattern-Fragment **PF** besitzen, bspw. für die Anwendung von Proxy, Factory oder Singleton Patterns. Bei der Realisierung von Filtern **FI** genügen **C** und **L**, da hier nur Code entfernt wird und so bspw. überflüssige Output-Statements oder Kommentare beseitigt werden können.

Durch die entsprechende Anwendung von **proxy patterns** können local calls in remote calls oder in ähnlicher Weise local classes in EJB-classes (Enterprise Java Beans) umgewandelt werden.

Umgekehrt kann mit Hilfe einer Transformation **T** und entsprechenden Regeln **TR** aus dem XML-codierten Quellcode **JavaML** oder einem Fragment dieses Codes auch ein gültiges Template **TP** generiert werden, das als Vorlage für anderen Quellcode anwendbar ist.

Die oben genannten Ausprägungen des erfindungsgemäßen Verfahrens können einzeln und in beliebiger Reihenfolge nacheinander erfolgen.

Durch das erfindungsgemäße Verfahren ergeben sich noch eine Reihe von zusätzlichen Vorteilen, wie beispielsweise:

1. Es können schnelle und flexible Änderungen im Quellcode vorgenommen werden.

2. Es ist nur ein System für Problemstellungen wie Patternanwendung, Migration, AOP, Filtering, etc. erforderlich und nicht eine Reihe verschiedener teilweise proprietärer Werkzeuge.

3. Das Verfahren basiert auf Standards wie XML und XSLT und ist hinsichtlich der Konvertierbarkeit in andere Programmiersprachen geringeren Beschränkungen unterworfen als andere Verfahren zur Modifikation von Quellcode.

4. Selbst für spezielle und komplizierte Quellcode-Modifikationen sind keine proprietären Speziallösungen erforderlich, sondern es können hierfür existierende Standards wie **XSLT**, **XPath** und **Xquery** genutzt werden.

5. Diese Art der Modifikation erlaubt die Erstellung von Hierarchien u.a. durch die Möglichkeit der Hintereinanderausführung (Pipelines) mehrerer Transformationen.

7. Die Transformationen können als allgemeine Transformationen für eine Wiederverwendung in XSLT-Dateien gespeichert werden, so daß Bibliotheken z.B. für bestimmte Abläufe entstehen können.

8. Es kann eine XML-Repräsentation des Quellcodes in einer XML-Datenbasis gespeichert und bei Bedarf mit Hilfe einer XSLT in einfacher Weise an die jeweiligen Kundenbedürfnisse angepasst werden.

9. Durch die Verwendung der Standards **XmlSchema** oder **DTD** oder entsprechende XSLTs kann der Code vorab (ohne Kompilierung), auf bestimmte Korrektheitsaspekte hin, überprüft (validiert) werden.

10. Übliche XML-Visualisierungstools Tools können zur einfachen Bearbeitung bzw. Visualisierung und Bestimmung von Beziehungen im Code verwendet werden.

11. Dauerhafte XML-basierte Programmbibliotheken, die XPath-Anfragen unterstützen, können die Wiederverwendung von Code durch besseres Auffinden eines Codes bzw. von Code-Fragmenten oder Templates verbessert werden.

4. Erfindungsvariante

In **Figur 8** ist ein Gesamtblockdiagramm zur Erläuterung der Erfindung dargestellt, bei dem zunächst ein Quellcode **SC**

durch einen Konverter **CONV** in einen in einer Meta-Auszeichnungssprache formulierten ersten Code **CodeML** umwandelt wird, wobei der Quellcode **SC** bei sofortiger Kompilierung einen Byte Code bzw. Binärcode **B** ergeben kann. Zu dem in der Meta-Auszeichnungssprache dargestellten Code **CodeML** wird nun eine zusätzliche Information **INFO** auf dem Wege einer durch Transformationsregeln **TR** beschriebenen Transformation **T** dem Code **CodeML** bzw. dem letztlich dem Quellcode **SC** hinzugefügt, wodurch sich ein zweiter ebenfalls in der Meta-Auszeichnungssprache formulierten Code **CodeML*** ergibt. Ein weiterer Konverter **RCONV** wandelt nach der Transformation den Code **CodeML*** in einen Quellcode **SC*** zurück, der typischerweise in derselben Sprache wie der Quellcode **SC** formuliert ist. Durch einen Compiler **COMP** wird schließlich der modifizierte Code **SC*** in einen modifizierten Byte Code **B*** oder aber gleich in einen ausführbaren Binärcode umgewandelt. Wesentlich ist hierbei, dass sich der Byte-Code **B*** vom Byte-Code **B** unterscheidet bzw. dass der Quellcode nicht nur in seiner Darstellung, sondern auch in seinem Programmablauf geändert wurde.

Der Quellcode **SC** und der modifizierte Quellcode **SC*** sind beispielsweise in der Programmiersprache Java und die Codes **CodeML** und **CodeML*** sind beispielsweise in der Meta-Auszeichnungssprache XML formuliert, wobei „XML" für Extended Markup Language steht.

In **Figur 10** ist ein erstes Ausführungsbeispiel dargestellt, bei dem die zusätzliche Information **INFO** in Form von Daten **D**, bspw. Initialisierungszuständen **SSDb**, Zustandsdaten **SDa**, Datenbankdaten **Dc**, beliebige Daten **x**, dem **CodeML** hinzugemischt werden, wobei diese Daten beispielsweise feste Zustände bzw. Werte für Konstanten und Variablen darstellen. Auf diese Weise kann der Quellcode **SC** mit festen Zuständen versorgt und transformiert werden, so dass ein benötigter Zustand zur Laufzeit des Programms, z.B. als Initialisierungszustand **SSDb**, sofort zur Verfügung steht und

nicht mehr gesondert ermittelt werden muss. Auf diese Weise können auch Objekt-Zustände in den Code eingebracht werden, die ein Wiederaufsetzen (Recovery) eines unterbrochenen Programms am selben Ort mit denselben Zuständen ermöglichen, ohne dass zusätzliche aufwändige programmtechnische Vorkehrungen hierfür getroffen werden müssen.

Die Transformation **T**, z. B. eine Extended Stylesheet Language Transformation oder **XSLT**, wird durch Transformationsregeln **TR**, z. B. innerhalb von **XSL** (Extended Stylesheet Language) Dateien beschrieben, wobei bspw. die in XSL formulierten Regeln u.a. beschreiben wie der in XML-codierte Quellcode **CodeML** mit den Zustandsdaten aus **D** kombiniert wird, um einen neuen modifizierten Quellcode **CodeML*** mit **SSDb**, **SDa** und **Dc** zu bilden.

Die Regeln einer Transformation **T** können dabei so geartet sein, dass die Informationen in ihrer ursprünglichen Form aber auch in einer durch Regeln geänderten Form hinzugemischt werden.

Die Regeln einer Transformation **T** können dabei auch so geartet sein, dass die Transformation **T**, z.B. mit Hilfe von if-conditions, durch die Informationen bzw. Daten beeinflusst werden.

Die im **Anhang 4** befindlichen Programmauflistungen **Listing 1** bis **Listing 6** zeigen dies an einem konkreten Beispiel, bei dem in einer Testklasse des Quellcodes die nicht initialisierte Variable **String m_sWelcome** in eine initialisierte Form **String m_sWelcome = "hello";** transformiert wird. Hierbei zeigt das **Listing 1** das entsprechende Java-Programm TestOutput.java, dass in eine XML-Darstellung TestOutput.xjava konvertiert wird. In **Listing 3** ist die XML-Datei State.XML mit dem Zustandswert **"hello"** dargestellt. In **Listing 4** folgt dann die Transformationsanweisung zum Vermischen **Mixing.xsl**, die mit

Anweisungen wie **template match** = und **apply-templates** dafür sorgen, dass der Code an der richtigen Stelle modifiziert wird.

In **Figur 9** ist ein zweites Ausführungsbeispiel dargestellt, bei dem ein in XML codiertes Code-Fragment **CFb** mit einem ursprünglichen in XML codierten Quellcode **CodeML**, der ein Code-Fragment **CFa** enthält, durch die Transformation **T** derart transformiert wird, dass im modifizierten XML-codierten Quellcode **CodeML*** anstelle des bisher vorhandenen Fragments **CFa** ein Code-Fragment **CFb** enthalten ist. Die Transformation **T** wird hierbei auch von Transformationsregeln **TR** gesteuert. Ein derartiger Austausch von Code-Fragmenten kann unter bestimmten Umständen z.B. als „Patching" bezeichnet werden. Durch das erfindungsgemäße Verfahren kann ein Patching in konsistenter Weise mit einem maximalen Grad an Freiheit für den Softwareentwickler erreicht werden, wobei dies beispielsweise automatisch und unter Berücksichtigung gegenseitiger Abhängigkeiten erfolgen kann.

Für dieses Ausführungsbeispiel ist ein konkreter Fall durch die **Listings 1A bis 6A** der im **Anhang 5** befindlichen Programmauflistungen gezeigt. Aus dem Java-Source-Code **TestOutput.java** wird wiederum ein **TextOutput.xjava** erzeugt. In **Listing 3A** kan man die Datei **CodeFragment.xml** erkennen, die ein Code-Fragment bereitstellt. Das **Listing 4A** enthält nun in der Datei **Patching.xsl** die Regeln für die Transformation **T**, wobei wiederum die Befehle **template match** = und **apply-templates** zum Einsatz kommen. Im **Listing 5A** ist dann der Inhalt der Datei **TestOutput.xjava(*)** mit dem modifizierten XML Quellcode und in **Listing 6A** in der Datei **TestOutput.java(*)** der modifizierte Java Quellcode dargestellt. Bei diesem Beispiel wird in der öffentlichen Testklasse die Stringzuweisung **String m_sWelcome = "hello";** durch eine Stringzuweisung **String m_sWelcome = System.getProperty("WELCOME");** ersetzt, wobei hier also der feste Wert "hello" durch die vom System angeforderte

Eigenschaft "WELCOME" ersetzt wird, und die beispielsweise irrtümlich „hardcodierte" Wertzuweisung nun „gepatched" werden kann.

Figur 11 betrifft ein drittes Anwendungsbeispiel der Erfindung, bei dem die Informationen **INFO** von **Zeichnung 1** nicht nur in der oben angegebenen Weise in Form von Informationen **INFO1**, sondern auch zusätzlich Form von in den Transformationsregeln eingebetteten Informationen **INFO2** bzw. Fragmenten hinzugemischt werden.

Figur 12 betrifft ein viertes Anwendungsbeispiel der Erfindung, bei dem XML-Quellcode **CodeML** mit dem Codefragment **CF**, das die Fremdsprachenfragmente **LF1** und **LF2** enthält, durch die Transformation **T** kombiniert wird, um einen modifizierten Code **CodeML***, beispielsweise angepasst an die natürliche Sprache des Benutzers (L1=german), zu erhalten. Die Transformation **XSLT** wird hierbei von Transformationsregeln **TR** bestimmt, die die zu ändernden Stellen des Quellcodes sowie die jeweilige ausgewählte natürliche Sprache, also beispielsweise **german** oder **english**, spezifiziert. Durch das erfindungsgemäße Verfahren wird so eine Lokalisierung und Internationalisierung des Quellcodes auf effiziente und ökonomische Weise, bei einer Minimierung der hierfür erforderlichen zusätzlichen Laufzeit, erreicht.

Die oben genannten Ausprägungen des erfindungsgemäßen Verfahrens können einzeln und in beliebiger Reihenfolge nacheinander erfolgen.

Durch das erfindungsgemäße Verfahren ergeben sich eine Reihe von Vorteilen, wie beispielsweise:

1. Es können schnelle und flexible Änderungen im Quellcode vorgenommen werden.

2. Es ist nur ein System für Problemstellungen wie Patching, Customizing, Updating, etc. erforderlich und nicht eine Reihe verschiedener teilweise proprietärer Werkzeuge.

3. Das Verfahren basiert auf Standards wie XML und XSLT und ist hinsichtlich der Konvertierbarkeit in andere Programmiersprachen geringeren Beschränkungen unterworfen als andere Verfahren zur Modifikation von Quellcode.

4. Selbst für spezielle und komplizierte Quellcode-Modifikationen sind keine proprietären Speziallösungen erforderlich, sondern es können hierfür existierende Standards wie **XSLT**, **XPath** und **XQuery** genutzt werden.

5. Diese Art der Modifikation erlaubt die Erstellung von Hierarchien u.a. durch die Möglichkeit zur geordneten, automatisierten Hintereinanderausführung (Pipelines) mehrerer Transformationen, bspw. von Patches.

7. Die Transformationen können für eine allgemeine Wiederverwendung in XSLT-Dateien gespeichert werden, so daß Bibliotheken z.B. für bestimmte Abläufe entstehen können.

8. Es kann eine XML-Repräsentation des Quellcodes in einer XML-Datenbasis gespeichert und bei Bedarf mit Hilfe einer XSTL in einfacher Weise an die jeweiligen Kundenbedürfnisse angepasst werden (Customization).

9. Durch die Verwendung der Standards **XMLSchema** oder **DTD** oder entsprechende XSLTs kann der Code vorab (ohne Kompilierung), auf bestimmte Korrektheitsaspekte hin, überprüft (validiert) werden.

10. Übliche XML-Tools können zur einfachen Bearbeitung bzw. Visualisierung und Bestimmung von Beziehungen im Code verwendet werden.

11. Dauerhafte XML-basierte Programmbibliotheken, die XPath-Anfragen unterstützen, können die Wiederverwendung von Code durch besseres Auffinden eines Codes bzw. von Code-Fragmenten oder Templates verbessert werden.

Anhang 1:

Listing 1: TestRegistry.java

```
import electric.registry.Registry;
public class TestRegistry
{
  … //weiterer Quellcode in dem etwas geändert werden muss
}
```

Listing 2: TestRegistry.xjava

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <import>
    <dot><dot><name>electric</name><name>registry</name></dot><name>Registry</name></dot>
  </import>
  <class>
    <modifiers><public/></modifiers>
    <name>TestRegistry</name>
    <block>
      …
    <block>
  </class>
</java>
```

Listing 3: VariationPointT1.xsl

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <!-- ********************************
       *** copy template   **
   ********************************-->
  <xsl:template match="*">
    <xsl:copy><xsl:apply-templates/></xsl:copy>
  </xsl:template>
  <!-- ********************************************
       ***  Variotion Point Transformation 1    **
   ********************************************-->
  <xsl:template match="import">
    <xsl:if test="dot/name='Registry'">
      <import>
        <dot>
          <dot><dot><dot><name>org</name><name>apache</name></dot><name>axis</name></dot>
          <name>client</name></dot><name>Call</name>
        </dot>
      </import>
      <import>
        <dot>
          <dot><dot><dot><name>org</name><name>apache</name></dot><name>axis</name></dot>
            <name>client</name></dot><name>Service</name>
        </dot>
      </import>
    </xsl:if>
    <xsl:if test="dot/name!='Registry'">
      <xsl:copy-of select="."/>
    </xsl:if>
    …
  </xsl:template>
</xsl:stylesheet>
```

Listing 4: TestRegistry.xjava (*)

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <import>
    <dot>
      <dot><dot><dot><name>org</name><name>apache</name></dot><name>axis</name></dot>
      <name>client</name></dot><name>Call</name>
    </dot>
  </import>
  <import>
    <dot>
      <dot><dot><dot><name>org</name><name>apache</name></dot><name>axis</name></dot>
        <name>client</name></dot><name>Service</name>
    </dot>
  </import>
  <class>
    <modifiers><public/></modifiers>
    <name>TestRegistry</name>
    <block>
      ...
    <block>
  </class>
</java>
```

Listing 5: TestRegistry.java (*)

```
import org.apache.axis.client.Call; //Axis
import org.apache.axis.client.Service;
public class TestRegistry
{
 ... //weiterer Quellcode in dem etwas geändert wurde
}
```

Anhang 2

Listing 1: TestOutput.xjava

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
   <define name="m" value="private">
  <class>
    <modifiers><m/></modifiers>
    <name>TestOutput</name>
    <block>
      <var>
        <type><name>String</name></type>
        <name>m_sWelcome</name>
      </var>
    </block>
  </class>
</java>
```

Listing 2: TestOutput.xjava*

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <class>
    <modifiers><private/></modifiers>
    <name>TestOutput</name>
    <block>
      <var>
        <type><name>String</name></type>
        <name>m_sWelcome</name>
      </var>
    </block>
  </class>
</java>
```

Listing 3: TestOutput.java

```
private class TestOutput
{
      String m_sWelcome;
}
```

Anhang 3

Listing 1: TestCalculator.java

```
public class TestCalculator{
   private int z;

   public void calculate(int x, int y){
       z = x+y;
   }
}
```

Listing 2: TestCalculator.xjava

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <class>
    <modifiers><public/></modifiers>
    <name>TestCalculator</name>
    <block>
      <var>
        <modifiers><private/></modifiers><type><int/></type><name>z</name>
      </var>
      <method>
        <modifiers><public/></modifiers>
        <type><void/></type>
        <name>calculate</name>
        <params>
          <param><type><int/></type><name>x</name></param>
          <param><type><int/></type><name>y</name></param>
        </params>
        <curly>
          <expr>
            <a>
              <name>z</name>
              <plus><name>x</name><name>y</name></plus>
            </a>
          </expr>
        </curly>
      </method>
    </block>
  </class>
</java>
```

Listing 3: LoggingAspect.xsl

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<!-- ********************************
     *** copy template   **
     ********************************-->
<xsl:template match="*">
   <xsl:copy><xsl:apply-templates/></xsl:copy>
</xsl:template>

<!-- ********************************
       ***  logging aspect            **
   ********************************-->
<xsl:template match="*[(name()='curly')and(ancestor::method[contains(name,'cal')])]">
  <xsl:copy>
   <expr>
    <paren>
     <dot><dot><name>System</name><name>out</name></dot><name>println</name></dot>
     <exprs>
      <expr>
```

```
          <xsl:text>"</xsl:text><xsl:value-of select="../name"/><xsl:text> begin"</xsl:text>
         </expr>
        </exprs>
       </paren>
      </expr>
      <xsl:copy-of select="*" />
      <expr>
       <paren>
        <dot><dot><name>System</name><name>out</name></dot><name>println</name></dot>
        <exprs>
         <expr>
          <xsl:text>"</xsl:text><xsl:value-of select="../name"/><xsl:text> end"</xsl:text>
         </expr>
        </exprs>
       </paren>
      </expr>
     </xsl:copy>
    </xsl:template>
</xsl:stylesheet>
```

Listing 4: TestCalculator*.xjava

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <class>
    <modifiers><public/></modifiers>
    <name>TestCalculator</name>
    <block>
      <var>
        <modifiers><private/></modifiers><type><int/></type><name>z</name>
      </var>
      <method>
        <modifiers><public/></modifiers>
        <type><void/></type>
        <name>calculate</name>
        <params>
          <param><type><int/></type><name>x</name></param>
          <param><type><int/></type><name>y</name></param>
        </params>
        <curly>
          <expr>
            <paren>
              <dot><dot><name>System</name><name>out</name></dot><name>println</name></dot>
              <exprs><expr>"calculate begin"</expr></exprs>
            </paren>
          </expr>
          <expr>
            <a>
              <name>z</name>
              <plus><name>x</name><name>y</name></plus>
            </a>
          </expr>
          <expr>
            <paren>
              <dot><dot><name>System</name><name>out</name></dot><name>println</name></dot>
              <exprs><expr>"calculate end"</expr></exprs>
            </paren>
          </expr>
        </curly>
      </method>
    </block>
  </class>
</java>
```

Listing 5: TestCalculator*.java

```
public class TestCalculator{
   private int z;

   public void calculate(int x, int y){
```

```
        System.out.println("calculate begin");
        z = x+y;
        System.out.println("calculate end");
    }
}
```

Anhang 4

Listing 1: TestOutput.java

```
public class TestOutput
{
      String m_sWelcome;
}
```

Listing 2: TestOutput.xjava

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <class>
    <modifiers><public/></modifiers>
    <name>TestOutput</name>
    <block>
      <var>
        <type><name>String</name></type>
        <name>m_sWelcome</name>
      </var>
    </block>
  </class>
</java>
```

Listing 3: State.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<state name="m_sWelcome">
  <value>"hello"</value>
</state>
```

Listing 4: Mixing.xsl

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<!-- ********************************
     *** copy template              **
     ********************************-->
<xsl:template match="*">
   <xsl:copy><xsl:apply-templates/></xsl:copy>
</xsl:template>
<!-- ********************************
     ***  mixing template          **
     ********************************-->
<xsl:template match="*[(name()='var')and(name='m_sWelcome')]">
 <xsl:copy>
   <xsl:copy-of select="*"/>
    <a>
     <expr><xsl:value-of select="//state[@name='m_sWelcome']/value" /></expr>
    </a>
  </xsl:copy>
</xsl:template>
```

```
</xsl:stylesheet>
```

Listing 5: TestOutput.xjava (*)

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <class>
    <modifiers><public/></modifiers>
    <name>TestOutput</name>
    <block>
      <var>
        <type><name>String</name></type>
        <name>m_sWelcome</name>
        <a>
          <expr>"hello"</expr>
        </a>
      </var>
    </block>
  </class>
</java>
```

Listing 6: TestOutput.java (*)

```
public class TestOutput
{
      String m_sWelcome="hello";
}
```

Anhang 5

Listing 1A: TestOutput.java

```
public class TestOutput
{
      String m_sWelcome="hello";
}
```

Listing 2A: TestOutput.xjava

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <class>
    <modifiers><public/></modifiers>
    <name>TestOutput</name>
    <block>
      <var>
        <type><name>String</name></type>
        <name>m_sWelcome</name>
        <a>
          <expr>"hello"</expr>
        </a>
      </var>
    </block>
  </class>
</java>
```

Listing 3A: CodeFragment.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<fragment name="m_sWelcome">
  <expr>
    <paren>
      <dot><name>System</name><name>getProperty</name></dot>
      <exprs><expr>"WELCOME"</expr></exprs>
    </paren>
  </expr>
</fragment>
```

Listing 4A: Patching.xsl

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <!-- ********************************
       *** copy template              **
       ********************************-->
  <xsl:template match="*">
    <xsl:copy><xsl:apply-templates/></xsl:copy>
  </xsl:template>
  <!-- ********************************
       ***  patching template         **
       ********************************-->
  <xsl:template match="*[(name()='a')and(ancestor::var/name='m_sWelcome')]">
```

```
      <xsl:copy>
        <xsl:copy-of select="//fragment[@name='m_sWelcome']/expr" />
      </xsl:copy>
    </xsl:template>
</xsl:stylesheet>
```

Listing 5A: TestOutput.xjava (*)

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <class>
    <modifiers><public/></modifiers>
    <name>TestOutput</name>
    <block>
      <var>
        <type><name>String</name></type>
        <name>m_sWelcome</name>
        <a>
          <expr>
            <paren>
              <dot><name>System</name><name>getProperty</name></dot>
              <exprs><expr>"WELCOME"</expr></exprs>
            </paren>
          </expr>
        </a>
      </var>
    </block>
  </class>
</java>
```

Listing 6A: TestOutput.java (*)

```
public class TestOutput
{
      String m_sWelcome=System.getProperty("WELCOME");
}
```

Patentansprüche

1. Verfahren zur Veränderung von Software,
- bei dem vorab herstellerseitig aus einer nur aus Quelltext (SC1, SC2, SC) bestehenden ursprünglichen Software eine Mischform der ursprünglichen Software derart gebildet wird, dass mindestens ein Teil (SC1) des Quelltexts in einen Byte- oder Binärcode (B1,..,B) compiliert und mindestens ein weiterer Teil (SC2) des Quelltexts in einen in einer Meta-Auszeichnungssprache formulierten Code (CodeML) für mindestens einen Variationspunkt (VP2,..VP) konvertiert wird,
- bei dem dann kundenseitig nach Bedarf nur mindestens ein Variationspunkt (VP2) der Mischform (SW) der ursprünglichen Software durch eine Transformation (T) in Abhängigkeit von Transformationsregeln (TR) in mindestens einen in der Meta-Auszeichnungssprache formulierten anderen Code (CodeML*) umgewandelt wird und
- bei dem der andere Code (CodeML*) direkt einen veränderten Variationspunkt (VP2*) einer angepassten Software (SW*) bildet oder aus dem anderen Code (CodeML*) durch einen Konverter (RCONV) ein Quellcode (SC*) und dann mittels eines Kompilers (COMP) ein Binär- oder ByteCode (B*) des veränderten Variationspunkts (VPB2) einer angepassten Software (SW*) gebildet wird, wobei sich die ursprüngliche (SW) und die angepasste Software (SW*) in ihrem Programmablauf bzw. Programminhalt unterscheiden.

2. Verfahren nach Anspruch 1,
bei dem die Transformationsregeln (TR) mindestens eine Modifikationsregel für einen Variationspunkt aufweisen.

3. Verfahren nach Anspruch 1 oder 2,
bei dem der Modifikationsregel ein Update auf eine neuere Softwareversion bzw. ein Patching anstößt.

4. Verfahren nach Anspruch 1 oder 2,

bei dem die Modifikation mindestens eines Variationspunktes (VP) durch die Transformation zur Laufzeit erfolgt.

5. Verfahren nach einem der vorhergehenden Ansprüche, bei dem die Programmiersprache des Quellcodes Java und die Meta-Auszeichnungssprache der Variationspunkte XML ist und bei dem die Transformation und die Regelbeschreibung mittels XSLT und XSL erfolgt.

6. Verfahren zur Veränderung von Quellcode,
- bei dem ein in einer Meta-Auszeichnungsprache formulierter erster Code (CodeML) mit mindestens in einer Meta-Auszeichnungssprache formulierten Spracherweiterungen (LE) als Ausgangscode zur Verfügung steht,
- bei dem der Ausgangscode durch eine Transformation (T) in Abhängigkeit von Transformationsregeln (TR) in einen in der Meta-Auszeichnungssprache formulierter zweiten Code (CodeML*) ohne die in der Meta-Auszeichnungssprache formulierten Spracherweiterungen (LE) überführt wird,
- bei dem die Transformationsregeln einen Sprachkonverter (LC) bilden, der die Spracherweiterungen (LE) des ersten Codes so auflöst bzw. anwendet, dass sie durch einen Rückkonverter (RCONV) der keine entsprechende Spracherweiterung aufweist verarbeitbar sind, und
- bei dem dieser zweite Code in einen in der ersten Programmiersprache oder einer anderen Programmiersprache formulierten zweiten Quellcode (SC*) umwandelbar ist, und einen gültigen Binärcode/ByteCode (B*) ergibt.

7. Verfahren nach Anspruch 6,

bei dem im zweiten Code (CodeML*) mindestens eine Spracherweiterung (LE*) neu erzeugt oder aus dem ersten Code (CodeML) übernommen wird, und diese Erzeugung oder Übernahme der Sprachkonverter (LC) vornimmt.

8. Verfahren zur Veränderung von Quellcode,
- bei dem ein in einer ersten Programmiersprache formulierter Quellcode (SC) in einen in einer Meta-Auszeichnungssprache formulierten ersten Code (CodeML) umgewandelt wird,
- bei dem eine Transformation (T) nur in Abhängigkeit von Transformationsregeln TR in einen in der Meta-Auszeichnungssprache formulierten zweiten Code (CodeML*) erfolgt und
- bei dem dieser zweite Code in einen in der ersten Programmiersprache oder einer anderen Programmiersprache formulierten zweiten Quellcode (SC*) verwandelt wird, wobei sich der erste und der zweite Quellcode in ihrer Funktionalität (B,B*) unterscheiden.

9. Verfahren nach Anspruch 8,
bei dem die Transformationsregeln (TR) mindestens eine Bedingung C und einen Logikbestandteil L und/oder Codefragment CF selbst enthalten.

10. Verfahren nach Anspruch 8 oder 9,
bei dem die Transformationsregeln (TR) mindestens ein Fragment in Form eines Templates (TPF) und/oder mindestens eines Patterns (PF) entalten, bei denen mit Hilfe der Transformation T mindestens eine Codeveränderung bewirkt.

11. Verfahren nach Anspruch 10,
bei dem TR derart ausgebildet ist, dass mit Hilfe der Transformationen T ein Mechanismus zur Sicherung mindestens eines Systemzustandes in den zweiten Quellcode (CodeML*) eingebracht wird, um eine Migration in andere Versionen zu ermöglichen.

12. Verfahren nach Anspruch 8 oder 9,
bei dem mit Hilfe der Transformation T aus dem ersten Code oder einem Fragment des ersten Codes mindestens ein Template (TP) gebildet wird.

13. Verfahren zur Veränderung von Quellcode,
- bei dem ein in einer ersten Programmiersprache formulierter Quellcode (SC) in einen in einer Meta-Auszeichnungssprache formulierten ersten Code (CodeML) umgewandelt wird,
- bei dem eine in der Meta-Auszeichnungssprache formulierte, den späteren Programmablauf (B*) beeinflussende Information (INFO) durch eine Transformation (T) zu diesem ersten Code ersetzend oder nichtersetzend hinzugefügt und so der ebenfalls in der Meta-Auszeichnungssprache formulierte zweiten Code (CodeML*) gebildet wird, wobei die Transformation in Abhängigkeit von in einer Transformationsbeschreibungssprache formulierten Transformationsregeln (TR) erfolgt, und
- bei dem dieser zweite Code in einen in der ersten Programmiersprache oder einer anderen Programmiersprache formulierten zweiten Quellcode (SC*) verwandelt wird, wobei sich der Programminhalt bzw. Programmablauf (B) des ersten Quellcodes (SC) vom Programminhalt bzw. Programmablauf (B*) des zweiten Quellcodes (SC*) unterscheidet.

14. Verfahren nach Anspruch 13,
bei dem diese Information (INFO) mindestens ein Code-Fragment (CFb) enthält und
bei dem der zweite Quellcode dadurch gebildet wird, dass mindestens ein im ersten Quellcode enthaltenes Code-Fragment (CFa) mit Hilfe der Transformation durch das mindestens eine im Fragment enthaltene Code-Fragment (CFb) ersetzt wird.

15. Verfahren nach Anspruch 13,

bei dem die Information (INFO) speziell Daten (D) in Form von Initialisierungszuständen (SSDb) oder Zustandsdaten (SDa) oder Datenbankdaten (Dc) enthalten.

16. Verfahren nach Anspruch 15,
bei die Transformationsregeln (TR) von diesen Daten (D) beeinflusst werden.

17. Verfahren nach einem der Ansprüche 13 bis 16,
bei dem Daten (D) und/oder Code-Fragmente (CF) zusätzlich in den Transformationsregeln eingebettet sind.

18. Verfahren nach einem der Ansprüche 13 bis 17,
bei dem von checkpoints generierte Daten mittels einer Transformation so hinzugefügt werden, das der interne Zustand des ursprünglichen Programms beim Neustart des Programms zur Verfügung steht bzw. von diesem genutzt werden kann.

19. Verfahren nach einem der Ansprüche 13 bis 17,
bei dem Informationen (INFO) Updates oder Patches enthalten.

20. Verfahren nach Anspruch 13 bis 17,
bei dem Fragmente (LF1, LF2) Informationen zur Internationalisierung enthalten, die der Anpassung an unterschiedliche natürliche Sprachen dienen.

21. Verfahren nach einem der Ansprüche 13 bis 17,
bei dem Daten und/oder Code-Fragmente aus einer Bibliothek entstammen und
auf Kunden oder Kundengruppen abgestimmte Informationen tragen.

22. Anordnung zur Veränderung von Software,
- bei der eine Mischform einer ursprünglichen Software derart vorhanden ist, dass mindestens ein Teil (SC1) eines Quelltexts in einen Byte- oder Binärcode (B1,..,B) compiliert und mindestens ein weiterer Teil (SC2) des Quelltexts in

einen in einer Meta-Auszeichnungssprache formulierten Code (CodeML) für mindestens einen Variationspunkt (VP2,..VP) konvertiert ist,
- bei der eine Einrichtung zur Transformation (T) derart vorhanden ist, dass nach Bedarf nur mindestens ein Variationspunkt (VP2) der Mischform (SW) der ursprünglichen Software durch die Transformation (T) in Abhängigkeit von Transformationsregeln (TR) in mindestens einen in der Meta-Auszeichnungssprache formulierten anderen Code (CodeML*) umwandelbar ist,
wobei der andere Code (CodeML*) direkt einen veränderten Variationspunkt (VP2*) einer angepassten Software (SW*) bildet oder aus dem anderen Code (CodeML*) durch einen Konverter (RCONV) ein Quellcode (SC*) und dann mittels eines Compilers (COMP) ein Binär- oder ByteCode (B*) des veränderten Variationspunkts (VPB2) einer angepassten Software (SW*) bildbar ist und wobei sich die ursprüngliche (SW) und die angepasste Software (SW*) in ihrem Programmablauf bzw. Programminhalt unterscheiden.

23. Anordnung zur Veränderung von Quellcode,
- bei der ein Prozessor derart vorhanden ist, dass eine Transformation (T) des Ausgangscodes (CodeML) in Abhängigkeit von in einer erweiterten Stilbeschreibungssprache formulierten Transformationsregeln (TR) und einen darin enthaltenen Sprachkonverter (LC) so durchgeführt wird, dass entweder ein in der Meta-Auszeichnungssprache formulierter zweiter Code (CodeML*) ohne die in der Meta-Auszeichnungssprache formulierten Spracherweiterungen (LE) des ersten Codes (CodeML),
oder ein in der Meta-Auszeichnungssprache formulierter zweiter Code (CodeML*) mit neuen und/oder den ursprünglichen in der Meta-Auszeichnungssprache formulierten Spracherweiterungen (LE) erzeugt wird, und
- bei der ein Rückkonverter (RCONV) derart vorhanden ist, dass dieser zweite Code in einen in der ersten

Programmiersprache oder einer anderen Programmiersprache formulierten Quellcode (SC*) verwandelt wird.

24. Anordnung zur Veränderung von Quellcode,
- bei der ein erster Konverter (CONV) derart vorhanden ist, dass ein in einer ersten Programmiersprache formulierter Quellcode (SC) in einen in einer Meta-Auszeichnungssprache formulierten ersten Code (CodeML) umgewandelt wird,
- bei der ein Prozessor derart vorhanden ist, dass der CodeML durch eine Transformation (T) nur in Abhängigkeit von Transformationsregeln (TR) in einen in der Meta-Auszeichnungssprache formulierten zweiten Code (CodeML*) umgewandelt wird und
- bei der ein zweiter Konverter (RCONV) derart vorhanden ist, dass dieser zweite Code in einen in der ersten Programmiersprache oder einer anderen Programmiersprache formulierten zweiten Quellcode (CodeML*) verwandelt wird, wobei sich der erste und der zweite Quellcode in ihrer Funktionalität bzw. Inhalt(B,B*) unterscheiden.

25. Anordnung zur Veränderung von Quellcode,
- bei der ein erster Konverter (CONV) derart vorhanden ist, dass ein in einer ersten Programmiersprache formulierter Quellcode (SC) in einen in einer Meta-Auszeichnungssprache formulierten ersten Code (CodeML) umgewandelt wird,
- bei der ein Prozessor derart vorhanden ist, dass eine in der Meta-Auszeichnungssprache formulierte, den späteren Programmablauf (B*) beeinflussende Information (INFO) durch eine Transformation (T) zu diesem ersten Code ersetzend oder nichtersetzend hinzugefügt und so der ebenfalls in der Meta-Auszeichnungssprache formulierte zweiten Code (CodeML*) gebildet wird, wobei die Transformation in Abhängigkeit von in einer Transformationsbeschreibungssprache formulierten Transformationsregeln (TR) erfolgt, und

- bei der ein zweiter Konverter (RCONV) derart vorhanden ist, dass dieser zweite Code in einen in der ersten Programmiersprache oder einer anderen Programmiersprache formulierten zweiten Quellcode (SC*) verwandelt wird, wobei sich der Programminhalt bzw. Programmablauf (B) des ersten Quellcodes (SC) vom Programminhalt bzw. Programmablauf (B*) des zweiten Quellcodes (SC*) unterscheidet.

FIG 1

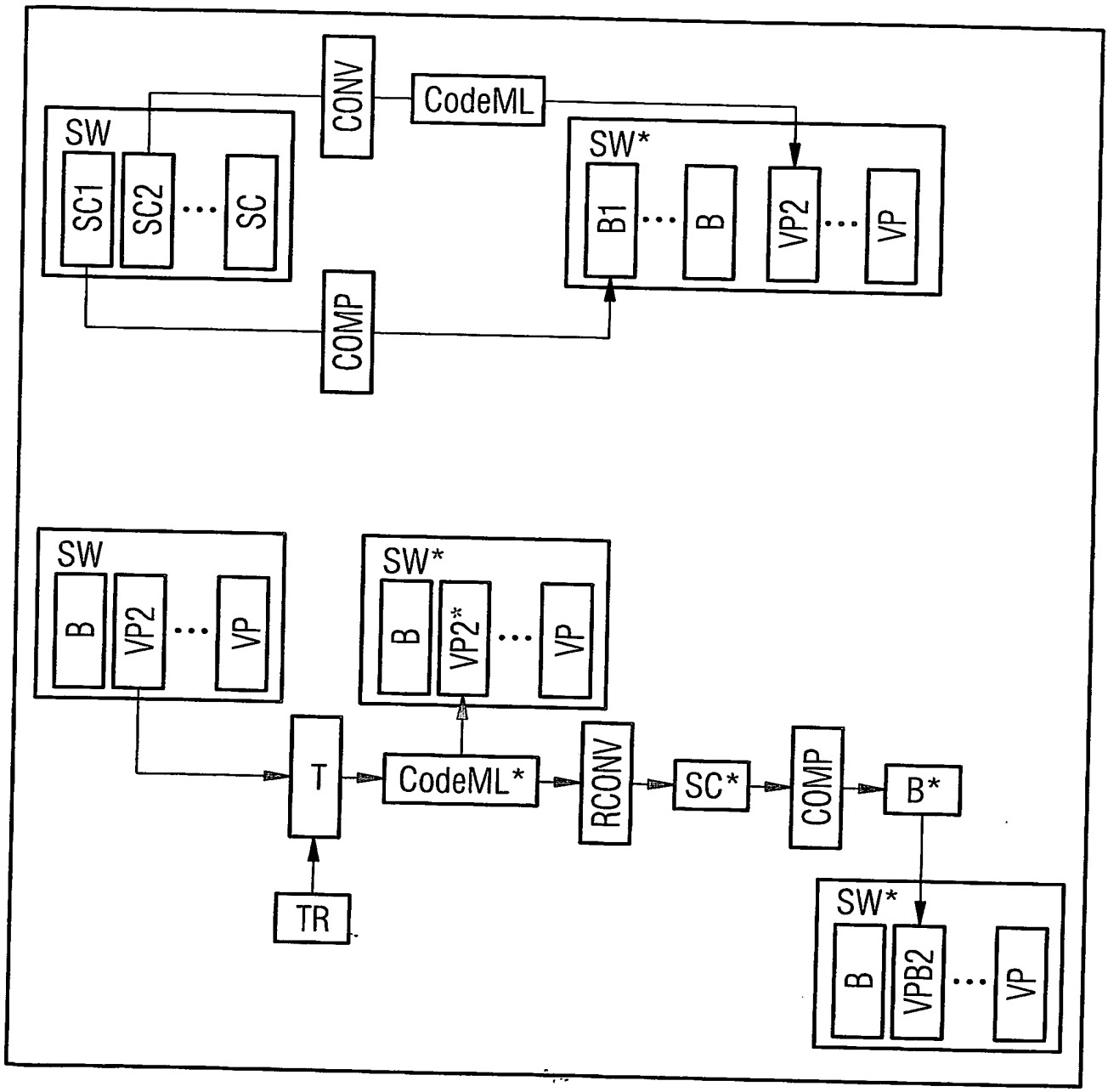

SW
SC1
SC2
SC
CONV
CodeML
SW*
B1
B
VP2
VP
COMP
SW
B
VP2
VP
SW*
B
VP2*
VP
T
CodeML*
RCONV
SC*
COMP
B*
TR
SW*
B
VPB2
VP

FIG 2

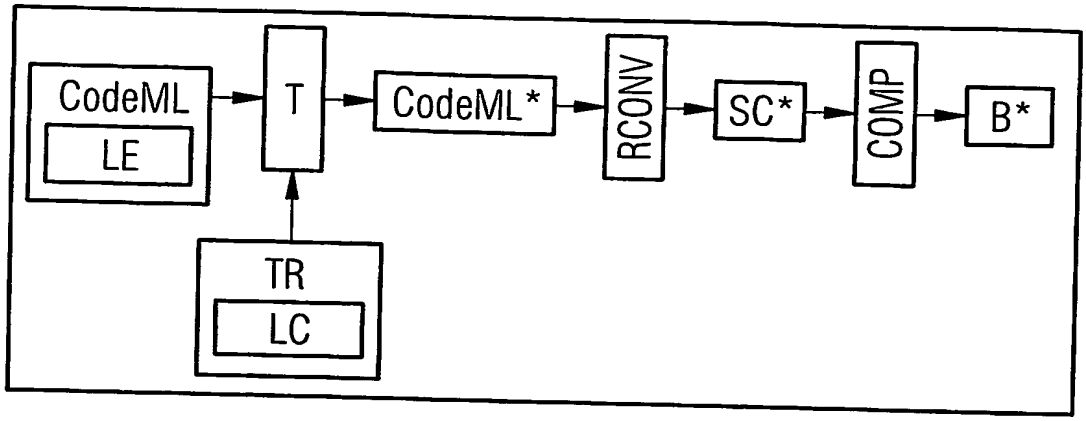


FIG 3

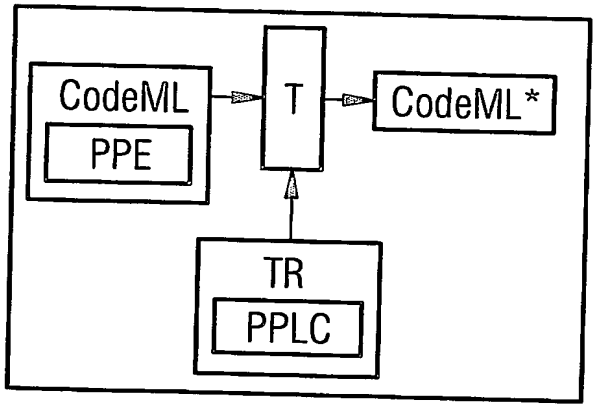


FIG 4

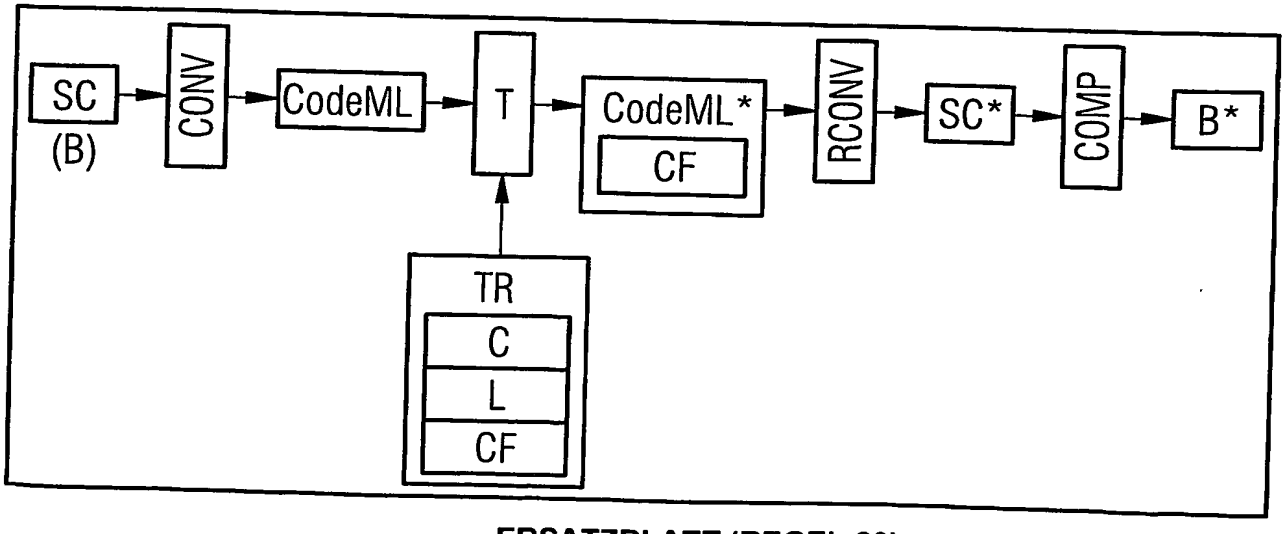


FIG 5

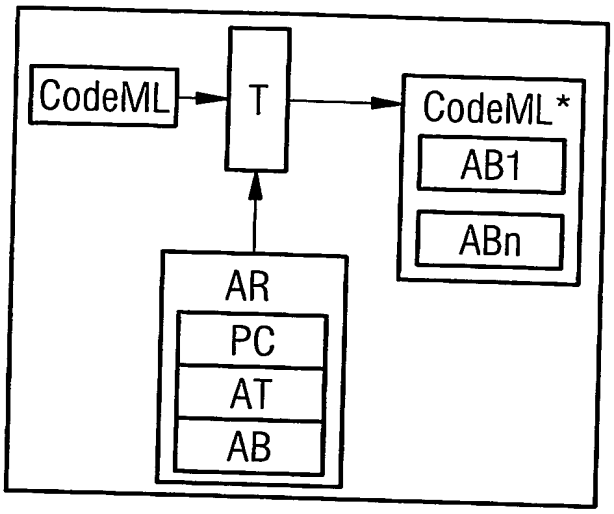

CodeML
T
CodeML*
AB1
ABn
AR
PC
AT
AB

FIG 6

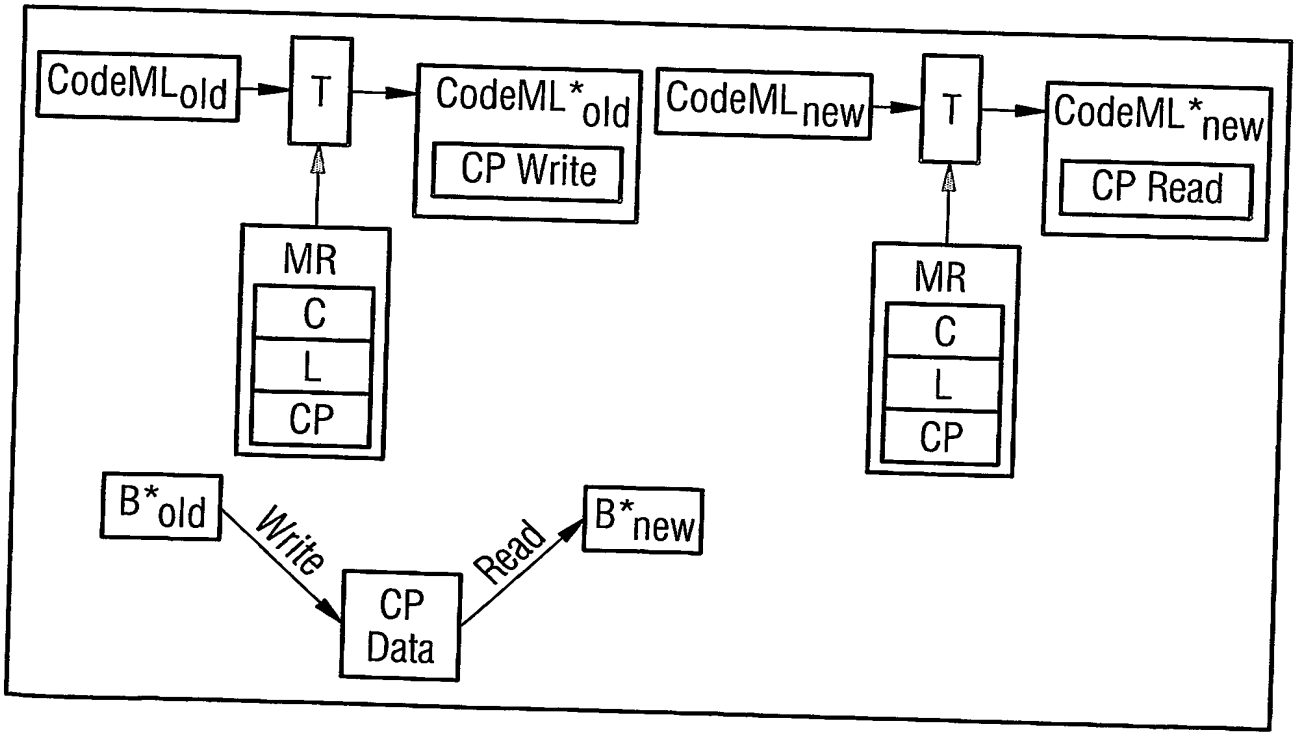

CodeMLold
T
CodeML*old
CP Write
CodeMLnew
T
CodeML*new
CP Read
MR
C
L
CP
MR
C
L
CP
B*old
Write
CP
Data
Read
B*new

FIG 7

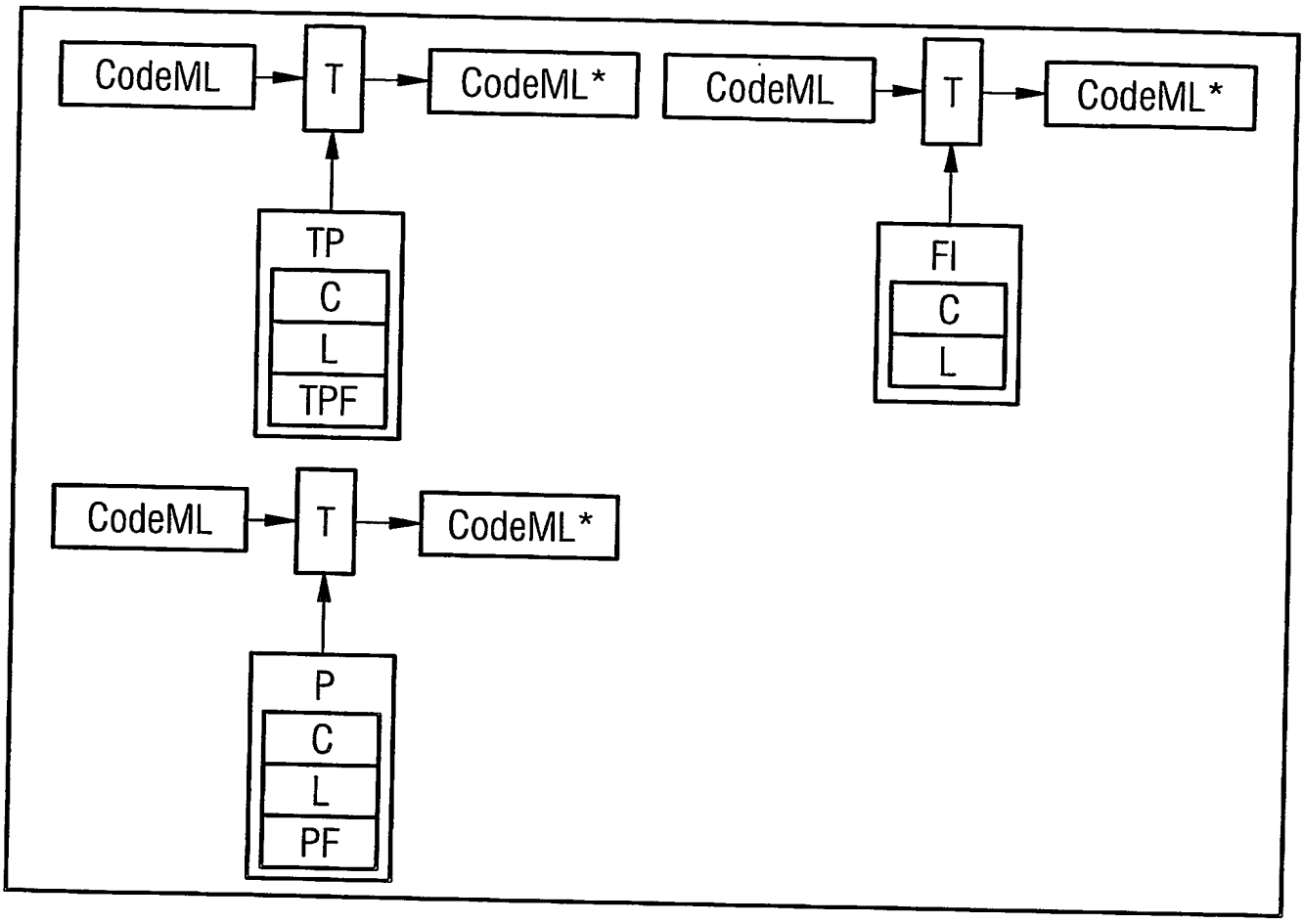

CodeML
T
CodeML*
TP
C
L
TPF
CodeML
T
CodeML*
FI
C
L
CodeML
T
CodeML*
P
C
L
PF

FIG 8

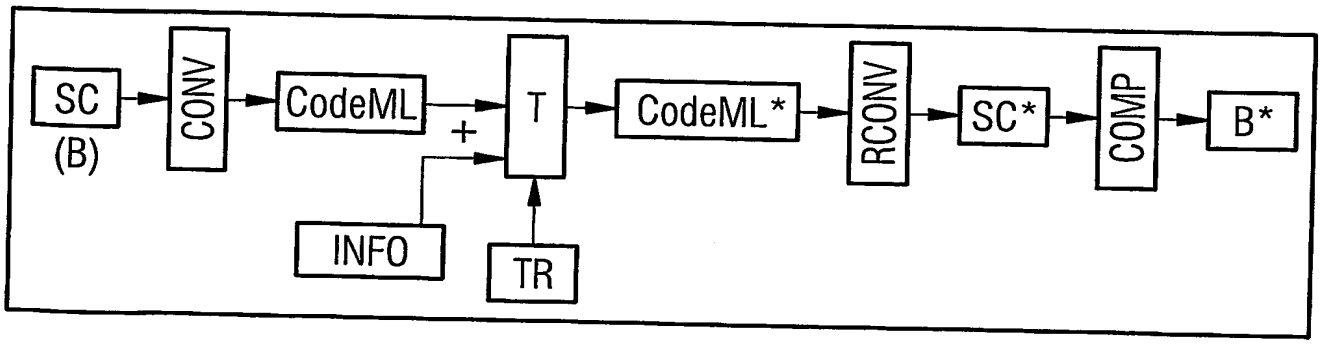

SC
(B)
CONV
CodeML
+
T
CodeML*
RCONV
SC*
COMP
B*
INFO
TR

FIG 9

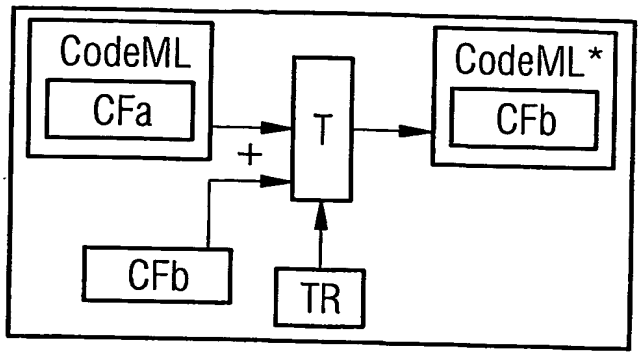


FIG 10

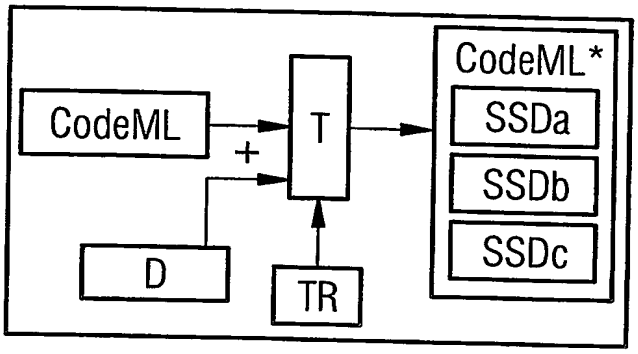


FIG 11

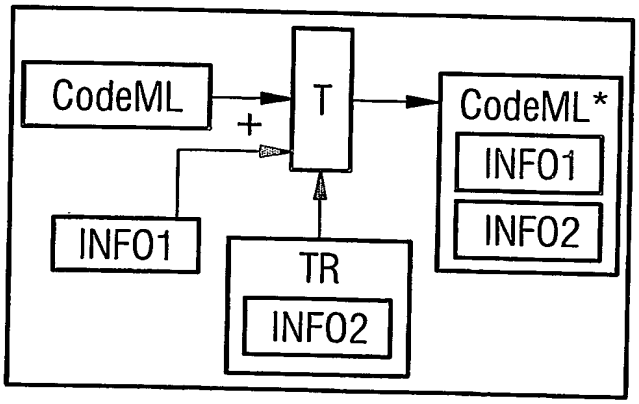


FIG 12